RDVIS: A Tool That Visualizes the Causes of Low Locality and Hints Program Optimizations

Kristof Beyls, Erik H. D'Hollander, and Frederik Vandeputte

Departement of Electronics and Information Systems,
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{kristof.beyls, erik.dhollander, frederik.vandeputte}@elis.ugent.be

Abstract. The visualization tool `rdvis` is presented which aims at helping the programmer to find program transformations to improve temporal data locality. We present a number of locality metrics that capture the necessary information. Based on a cluster analysis of basic block vectors, the tool gives strong hints on which program transformations are needed. The visualizer allowed us to find the necessary transformations for three SPEC2000 programs in just a few minutes. After performing these transformations, the programs run 3 times faster on average on a number of different platforms.

1 Introduction

For many applications, half of their execution time the processor performs no useful work due to data cache misses. Since the majority of the cache misses result from poor temporal or spatial locality, this bottleneck can be removed by increasing locality. In this paper, we propose a visualization of the causes of low locality. In contrast to earlier visualizations of cache misses[1, 4, 8, 10, 11, 13, 15, 16, 17], the proposed visualization is expected to give more useful hints on how to transform the code to improve locality and eliminate cache misses in a platform-independent way.

Whenever a memory location is used multiple times throughout the program execution (i.e. it is *reused*), the corresponding data may stay in the cache between the different accesses to it, and cache hits may result. However, when the reuses are separated by accesses to a lot of different data, the probability that the data remains in the cache between use and reuse is low. The number of different memory locations accessed between use and reuse is called the *reuse distance*[3].

The proposed visualization indicates the constructs in the source code that induce these low-locality reuses. By also highlighting the code that is executed between use and reuse, the programmer sees which code is responsible for increasing the reuse distance of a given pair of references. As such, the visualization shows the code that should be restructured to decrease reuse distances.

In a previous paper [2], we showed that this information helps the programmer to optimize programs platform-independently with an average speedup of 3 and a maximum of 11. However, the visualization was not as extensive as described in this paper, and more than a day of work was needed to understand each application's bottlenecks. With the visualization presented here, the application's bottlenecks can be understood in just a few minutes. In comparison to earlier work, we present the following novelties:

- The GCC compiler has been adapted to instrument programs to measure reuse distance and the code that is executed between reuses and to record the exact locations in the source code that lead to memory accesses.
- An automatic visualizer has been developed.
- The visualization might show a lot of different reference pairs that lead to long distance reuses. However, we found that many different long reference pairs can often be optimized by a single program transformation. Previously, it was unclear which reference pairs could be optimized by the same program transformation. In this paper, we propose a clustering algorithm, based on basic block vectors (BBV). In contrast to other work using BBVs [14] where BBVs indicate the basic blocks executed in a fixed-length interval of a program execution, our BBVs represent the code executed between data reuses.

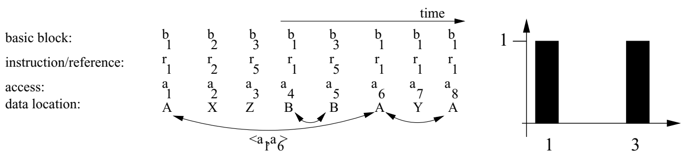
In Section 2, the proposed method for improving locality is presented. Section 3 presents the visualizations to support the programmer in effectively following the method. Examples of the visualization are discussed in Section 4, and concluding remarks follow in Section 5.

2 Method for Improving Data Locality

In this section, the visualization-based method presented in [2] to improve data locality is briefly presented. It focuses on resolving capacity misses, the dominant kind of misses for most applications and cache configurations [5].

Quantifying Poor Locality. A capacity miss occurs when too many different memory locations are accessed between reuses of the same data. This is described more formally in the definition and lemma below.

Definition 1. *A* **memory reference** *corresponds to a read or a write in the source code, while a particular execution of that read or write at runtime is a* **memory access***[7]. A* **reuse pair** $\langle a_1, a_2 \rangle$ *is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The* **reuse distance** *of a reuse pair* $\langle a_1, a_2 \rangle$ *is the number of unique memory locations accessed between accesses* a_1 *and* a_2*. A* **reference pair** $\langle r_1, r_2 \rangle$ *is a pair of memory references. The histogram of all reuse distances of the reuse pairs* $\langle a_1, a_2 \rangle$*, for which* a_1 *is an access produced by* r_1*, and* a_2 *is produced by* r_2*, is called the* **reference pair's reuse distance histogram RDH($\langle r_1, r_2 \rangle$)***. The* **basic block vector of a reference pair** $\langle r_1, r_2 \rangle$ **BBV($\langle r_1, r_2 \rangle$)** *is a vector which contains real numbers between 0 and 1, each corresponding to a basic block in the program. The value of an element is the fraction of reuse pairs in* $\langle r_1, r_2 \rangle$ *for which the basic block is executed between use and reuse. (see Fig. 1)* □


time
basic block:
instruction/reference:
access:
data location:
b_1 b_2 b_3 b_1 b_3 b_1 b_1 b_1
r_1 r_2 r_5 r_1 r_5 r_1 r_1 r_1
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
A X Z B B A Y A
$<a_1,a_6>$
1
1
3

b_1	b_2	b_3
1.0	0.5	0.5

(a) access stream (b) RDH($\langle r_1, r_1\rangle$) (c) BBV($\langle r_1, r_1\rangle$)

Fig. 1. Example memory access stream. In (a), the reuse pairs are indicated by arrows. The second row indicates 8 memory instructions, generating the memory accesses in the third row. The top row indicates the basic blocks containing the references in the second row. The bottom row shows the corresponding addresses. The reuse distance of $\langle a_1, a_6\rangle = 3$. (b) shows RDH($\langle r_1, r_1\rangle$), while (c) shows BBV($\langle r_1, r_1\rangle$)

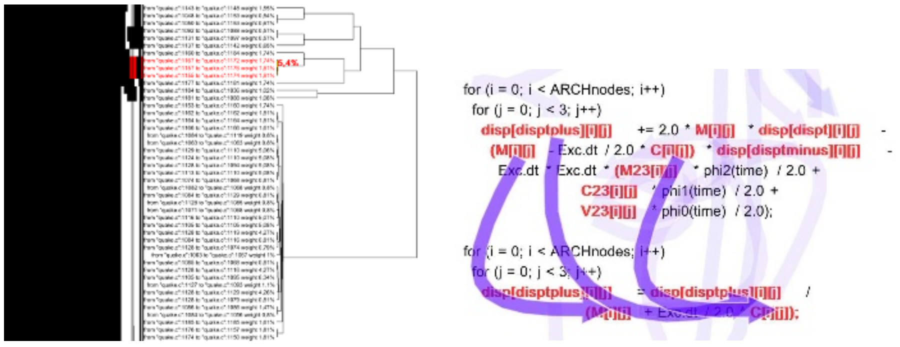
Lemma 1. *In a fully associative LRU cache with n lines, a reuse pair $\langle a_1, a_2\rangle$ with reuse distance $d < n$ will generate a hit at access a_2. If $d \geq n$, a_2 misses the cache.* □

Program Transformations to Improve Data Locality. The locality measurement and visualization is performed automatically. Based on the visualization, the capacity misses can be minimized by the programmer. The most important way to eliminate capacity misses is to reduce the distance between use and reuse for long reuse distances, so that it becomes smaller than the cache size. This can be done by reordering computations (and memory accesses), so that *higher temporal locality* is achieved. Examples of such optimizations are loop fusion and loop tiling[12]. The general idea behind this family of optimizations is to try to do as much useful computations as possible on the data while it is in the cache.

3 Support Provided by the Rdvis Tool

3.1 Instrumentation

The reference pairs, their reuse distance histogram and the code executed between use and reuse is measured by instrumenting the program. The instrumentation is implemented in the GCC compiler. The C front end has been adapted to record the precise location of memory references in the source code. The instrumentation occurs in an early stage of the compilation process so that the source code information is still available. All the pointer dereferences and array accesses are assumed to lead to load and store instructions in the resulting binary. It is assumed that all scalar variables can be allocated to registers, so they don't lead to memory accesses. Even when the register file of the processor is too small to contain all scalar variables, the accesses are very likely to have very good locality, so it is not necessary to take them into account. At the end of program execution all the recorded reuse pairs, the associated distance histograms and BBVs of intermediately executed code are written to a file.



```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j]     += 2.0 * M[i][j]  * disp[dispt][i][j]   -
      (M[i][j]  - Exc.dt / 2.0 * C[i][j])  * disp[dispminus][i][j]   -
        Exc.dt * Exc.dt * (M23[i][j]   * phi2(time)  / 2.0 +
                      C23[i][j]    * phi1(time)  / 2.0 +
                      V23[i][j]    * phi0(time)  / 2.0);

for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j]     = disp[disptplus][i][j]    /
                 (M[i][j]  + Exc.dt / 2.0 * C[i][j]);
```

(a) Cluster analysis view of Equake. A cluster of three reference pairs has been selected, which together produce 5.4% of all capacity misses. As a result, the corresponding arrows are highlighted in Fig. 2(b). The same clustering, with a different selected cluster is shown zoomed-in in Fig. 4

(b) Since the only highlighted code is in the two consecutive loops, it can be concluded that the use is in the first loop nest and the reuse is in the consecutive execution of the second loop nest. Therefore, the use and reuse can be brought closer together by loop fusion

Fig. 2. Cluster of reference pairs that can be optimized using loop fusion

3.2 Visualization

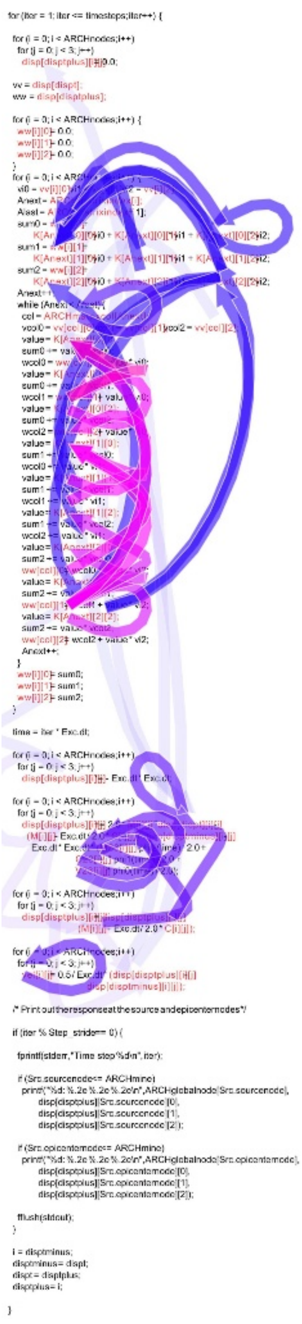
The visualizer is written in Java and reads the recorded reuse information. The following paragraphs describe the two different kinds of visualization.

Reference Pair Visualization. The reference pairs are visualized by drawing an arrow from the first reference to the second reference in the source code. Each arrow has a weight, which is the percentage of capacity misses induced by the corresponding reference pair. Only the heaviest arrows are drawn, so that the drawn arrows together represent at least 90% of all capacity misses. An example of such arrows are found in Figures 2(b) and 3. The color of the arrows corresponds to their weight, ranging from red for heavy-weighted arrows to blue for light-weighted arrows. The transparency of the arrows allows to better see the code covered by the arrows. Only the arrows that are of interest (see cluster analysis below) are clearly visible. The other arrows are made highly transparent, to make them nearly invisible, so that the users attention is not disturbed by the arrows that are not currently under investigation.

When the arrow is clicked by the mouse a pop-up menu lets the user choose between showing the reuse distance histogram of that reference pair, or highlighting the memory references that are executed between the use and the reuse. An example of such highlighted memory references can be found in Figures 2(b) and 3.

Clustering of Reference Pairs with Large Reuse Distances. After seeing the reuses at long distances, the programmer needs to think about how to reduce the reuse distance. He can do this by using program transformations that reduce the amount of data accessed between use and reuse. For the programs that we visualized, we found that typically one or two program transformations improve the majority of long-distance reuses, even though up to 30 different reference pairs are found in the 90th percentile of long distance-reuses. So, different reference pairs can be optimized by the same program transformation. Since the programmer should focus on the needed program transformations, and not on individual reference pairs, the visualizer clusters the pairs, so that those in the same cluster are likely to be resolved by the same program transformation.

The basic assumption in the clustering of reference pairs is that to diminish the long distance reuses, memory accesses executed between use and reuse should be moved either before the use or after the reuse. When different reference pairs have the same code executed between use and reuse, they are very likely to need the same program transformation to have less data accessed between the reuses. Therefore, for each reference pair, the basic block vector is recorded, which represents the code that is executed between use and reuse. Then, these BBVs are clustered, using a hierarchical agglomerative clustering algorithm[9], where the distance between two basic block vectors is determined by the Manhattan metric[14]. Examples of the visual representation of the computed hierarchical clusters are shown in Figures 2(a) and 4. In the middle, a textual description of each reference pair is shown. On the right a dendrogram is shown which graphically indicates the distance between subclusters. On the left, the basic block vectors for each reference pair is shown. A black pixel means that the corresponding basic block is never executed between use and reuse, while a white pixel indicates that the corresponding block is always executed between use and reuse. Based on this visualization, the user selects interesting subclusters, by clicking a subcluster in the dendrogram shown on the right hand side.



```
for (iter = 1; iter <= timesteps;iter++) {

  for (i = 0; i < ARCHnodes;i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] = 0.0;

  vv = disp[dispt];
  ww = disp[disptplus];

  for (i = 0; i < ARCHnodes;i++) {
    ww[i][0] = 0.0;
    ww[i][1] = 0.0;
    ww[i][2] = 0.0;
  }
  for (i = 0; i < AR[illegible]
    vi0 = vv[i][0]; vi1 [illegible] 2 = vv[i][2];
    Anext = AR[illegible]ndex[i];
    Alast = A[illegible]xindex[i + 1];
    sum0 = [illegible]
        K[Ar[illegible]0][0]*vi0 + K[A[illegible]xt][0][1]*vi1 + K[A[illegible]xt][0][2]*vi2;
    sum1 = ww[i][1] +
        K[Anext][1][0]*vi0 + K[A[illegible]xt][1][1]*vi1 + K[A[illegible]xt][1][2]*vi2;
    sum2 = ww[i][2] +
        K[Anext][2][0]*vi0 + K[Anext][2][1]*[illegible]xt][2][2]*vi2;
    Anext++;
    while (Anext < Alast) {
      col = ARCHm[illegible]col[Anext];
      vcol0 = vv[col][0[illegible][1]; vcol2 = vv[col][2];
      value = K[Anext][[illegible]
      sum0 += val[illegible]
      wcol0 = ww[col[illegible]value * vi0;
      value = K[Anext][[illegible]
      sum0 += [illegible]
      wcol1 = w[illegible]1] + valu[illegible]vi0;
      value = K[A[illegible]][0][2];
      sum0 += value * vcol2;
      wcol2 = [illegible]][2] + value * [illegible]
      value = [illegible]ext][1][0];
      sum1 += value * vcol0;
      wcol0 += value * vi[illegible]
      value = K[Anext][1][1];
      sum1 += value * vcol1;
      wcol1 += value * vi1;
      value = K[Anext][1][2];
      sum1 += value * vcol2;
      wcol2 += value * vi1;
      value = K[Anext][2][[illegible]
      sum2 += val[illegible]
      ww[col][0] = wcol0 [illegible] vi2;
      value = K[Anext[illegible]
      sum2 += va[illegible]
      ww[col][1] = wcol1 + value * vi2;
      value = K[Anext][2][2];
      sum2 += value * vcol2;
      ww[col][2] = wcol2 + value * vi2;
      Anext++;
    }
    ww[i][0] = sum0;
    ww[i][1] = sum1;
    ww[i][2] = sum2;
  }

  time = iter * Exc.dt;

  for (i = 0; i < ARCHnodes;i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] = - Exc.dt * Exc.dt;

  for (i = 0; i < ARCHnodes;i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] = 2.0 [illegible] disp[dispt][i][j]
        (M[i][j] - Exc.dt / 2.0 [illegible]minus][i][j]
          Exc.dt * Exc.dt [illegible][i][j] [illegible](time) * 2.0 +
                        [illegible][i][j] phi1(time) * 2.0 +
                        [illegible][i][j] phi0(time) * 2.0);

  for (i = 0; i < ARCHnodes;i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] = disp[disptplus][i][j] /
                        (M[i][j] + Exc.dt / 2.0 * C[i][j]);

  for (i = 0; i < ARCHnodes;i++)
    for (j = 0; j < 3; j++)
      vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] -
                        disp[disptminus][i][j]);

  /* Print out the response at the source and epicenter nodes */

  if (iter % Step_stride == 0) {

    fprintf(stderr,"Time step %d\n", iter);

    if (Src.sourcenode <= ARCHmine)
      printf("%d: %.2e %.2e %.2e\n",ARCHglobalnode[Src.sourcenode],
          disp[disptplus][Src.sourcenode][0],
          disp[disptplus][Src.sourcenode][1],
          disp[disptplus][Src.sourcenode][2]);

    if (Src.epicenternode <= ARCHmine)
      printf("%d: %.2e %.2e %.2e\n",ARCHglobalnode[Src.epicenternode],
          disp[disptplus][Src.epicenternode][0],
          disp[disptplus][Src.epicenternode][1],
          disp[disptplus][Src.epicenternode][2]);

    fflush(stdout);
  }

  i = disptminus;
  disptminus = dispt;
  dispt = disptplus;
  disptplus = i;

}
```

Fig. 3. Highlighted arrows corresponding to the selected cluster in Fig. 4

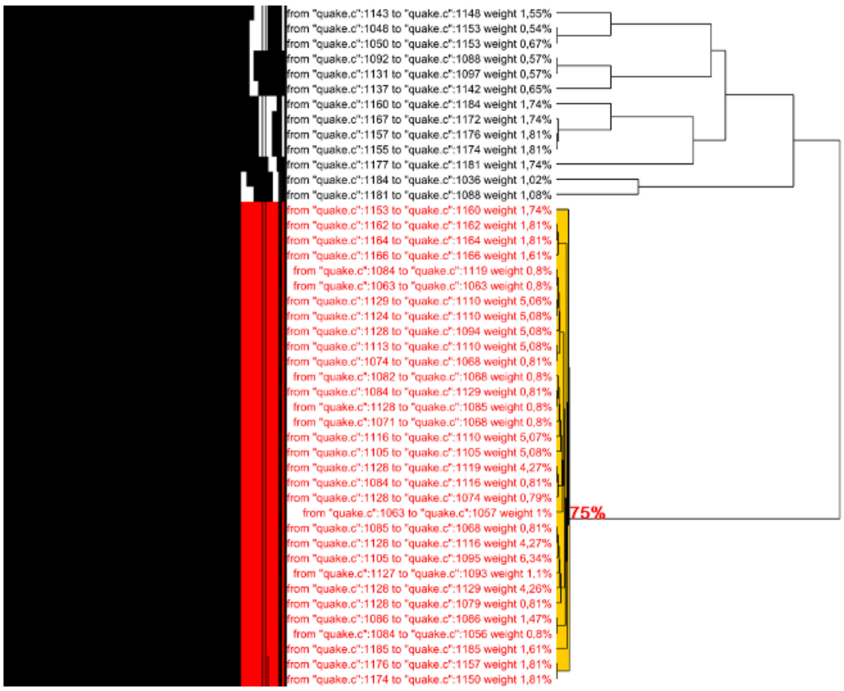


Fig. 4. The main cluster of Equake. The corresponding arrows and intermediately executed code is shown in figure 3

After selecting a subcluster the corresponding BBVs are highlighted, the percentage of long distance reuses generated by that cluster is shown in the dendrogram, and in the source code window, the corresponding arrows are highlighted, together with the memory references that are executed between uses and reuses. The other arrows are made highly transparent, so that they don't distract the users attention away from the currently investigated cluster.

4 Experiments

In [2], the three SPEC2000 programs with the largest cache bottleneck (Mcf, Equake, Art) were optimized, based on a manual drawing of the arrows corresponding to the most important reference pairs [2]. After manually looking for code that is executed between use and reuse, which took more than a day of work, program transformations were found to reduce the number of long reuse distances in Art and Equake. Mcf proved too difficult to analyze by hand, and only prefetch instructions could be inserted to try to hide the cache miss latency. As can be seen in Fig. 5, the optimized versions of Art and Equake, where the number of long reuse distances was reduced, gives consistent speed-up on multiple platforms. However, inserting prefetch instructions in Mcf does not reduce cache misses, and the optimization is not profitable on all platforms. In the rest of this section, we show how the tool can help identify the needed program transformations in just a couple of minutes.

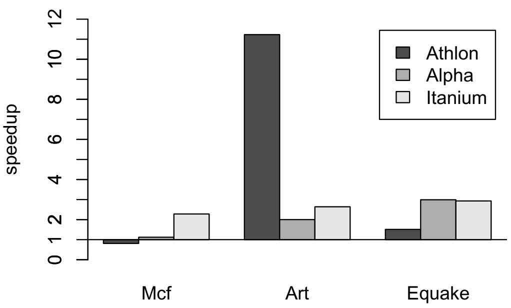


Fig. 5. Speedups after program transformations based on visualization of reference pairs

Loop Fusion. Based on the cluster analysis of the Equake program, as seen in Fig. 2(a), one can select the highlighted cluster in the figure, to see how this cluster of reference pairs could be optimized. When the cluster is selected, the corresponding arrows are highlighted, as well as the code that is executed between the uses and reuses that are represented by these arrows. From the visualization (see Fig. 2(b)), it can be concluded that loop fusion will remove these long-distance reuses, removing 5.4% of the capacity misses.

Loop Tiling. When selecting the cluster displayed in Figure 4, one sees that 75% of all long distance reuses are caused by this cluster. Figure 3 shows that between use and reuse, most of the memory references in the outer loop are executed. Therefore, the reuse occurs between different iterations of this outer loop, and the reuses can be brought together by tiling the loop.

5 Conclusions and Future Work

We have presented locality metrics (reuse distance, reference pairs and associated basic block vectors) that capture the necessary information to optimize the locality of programs. These metrics are measured by instrumenting the program and visualized after executing the program. The visualization is unique in the aspect of clustering the source code locations that lead to cache misses, so that each cluster needs a single program transformation to remove these cache misses. This enables the programmer to find the necessary program transformations to improve locality in a time-scale of minutes, whereas previously this could take days. The visualization proofs effective, since the three SPEC2000 programs with the largest memory bottlenecks could be optimized by it, leading to a three-fold speedup on average on a number of different platforms.

In the future, we would like to extend the presented work in a number of ways. First, we are thinking about using program slicing methods to automatically find independent code that is executed between use and reuse. Secondly, the possibility for predicting reuse distance distributions of long program runs, based on a short instrumented execution with small inputs[6], will be investigated. Thirdly, we are thinking about cluster analysis methods to guide data layout optimizations to improve spatial locality.

The original and optimized programs from the experiments are available at http://www.elis.UGent.be/~kbeyls. We plan to make the rdvis tool and the instrumentation patch for GCC available at the same place, after GCC 4.0 is released.

References

1. E. Berg and E. Hagersten. Sip: Performance tuning through source code interdependence. In *Euro-Par'02*, volume 2400 of *LNCS*, pages 177–186, 2002.
2. K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *ICCS 2004: 4th International Conference on computational science*, volume 3038 of *LNCS*, pages 448–455, 2004.
3. K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001.
4. R. Bosch and C. S. et al. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34(1):68–73, Feb. 2000.
5. J. F. Cantin and M. D. Hill. Cache performance for selected SPEC CPU2000 benchmarks. *Computer Architecture News (CAN)*, September 2001.
6. C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI'03*, ACM SIGPLAN Notices, 2003.
7. S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour.* PhD thesis, Princeton University, November 1999.
8. A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, 1993.
9. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
10. A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.
11. M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance in sequential and parallel programs. *IEEE Computer*, April 1995.
12. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
13. J. Mellor-Crummey and R. F. et al. HPCView: a tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–104, 2002.
14. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ASPLOS, SIGPLAN Notices*, 37(10):45–57, 2002.
15. E. Vanderdeijl, O. Temam, E. Granston, and G. Kanbier. The cache visualization tool. *IEEE Computer*, 30(7):71, 1997.
16. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on computational science*, volume 3038 of *LNCS*, pages 440–447, 2004.
17. Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of cache on the program execution. In *Information Visualization 2001*, pages 336–341, 2001.